Task-Based Polar Decomposition Using SLATE on Massively Parallel Systems with Hardware Accelerators

Dalal Sukkari
sukkari@icl.utk.edu
Innovative Computing Laboratory, University of Tennessee
Knoxville, TN, USA

Mark Gates
mgates3@icl.utk.edu
Innovative Computing Laboratory, University of Tennessee
Knoxville, TN, USA

Mohammed Al Farhan
mohammed.farhan@kaust.edu.sa
KAUST
Thuwal, Saudi Arabia

Hartwig Anzt
Jack Dongarra
hanzt@icl.utk.edu
dongarra@icl.utk.edu

ABSTRACT

We investigate a new task-based implementation of the polar decomposition on massively parallel systems augmented with multiple GPUs using SLATE. We implement the iterative QR Dynamically-Weighted Halley (QDWH) algorithm, whose building blocks mainly consist of compute-bound matrix operations, allowing for high levels of parallelism to be exploited on various hardware architectures, such as NVIDIA, AMD, and Intel GPU-based systems. To achieve both performance and portability, we implement our QDWH-based polar decomposition in the SLATE library, which uses efficient techniques in dense linear algebra, such as 2D block cyclic data distribution and communication-avoiding algorithms, as well as modern parallel programming approaches, such as dynamic scheduling and communication overlapping, and uses OpenMP tasks to track data dependencies.

We report numerical accuracy and performance results. The benchmarking campaign reveals up to an 18-fold performance speedup of the GPU accelerated implementation compared to the existing state-of-the-art implementation for the polar decomposition.

CCS CONCEPTS

• **Mathematics of computing → Solvers**; **Mathematical software performance**.

KEYWORDS

Linear algebra, polar decomposition, QDWH

ACM Reference Format:
Dalal Sukkari, Mark Gates, Mohammed Al Farhan, Hartwig Anzt, and Jack Dongarra. 2023. Task-Based Polar Decomposition Using SLATE on Massively Parallel Systems with Hardware Accelerators. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3624062.3624248

1 INTRODUCTION

With today's move to exascale computing, traditional numerical algorithms confront severe challenges, mainly due to the increased cost of communication relative to computation. Leveraging new algorithms geared to efficiently take advantage of the extreme levels of concurrency in exascale systems becomes mission-critical for pushing the edge of what is possible for science and engineering. This redesign comes with a cost – algorithms tailored for extreme parallelism, at the cost of extra floating-point operations.

This paper investigates the performance of a novel task-based implementation of the polar decomposition (PD) based on the iterative QR-based Dynamically Weighted Halley (QDWH) iteration on GPU-accelerated supercomputers. The polar decomposition of a matrix $A \in \mathbb{C}^{m\times n}$ $(m \geq n)$ is

$$A = U_p H,$$

where U_p is a unitary matrix and $H = \sqrt{A^\top A}$ is a symmetric positive semidefinite matrix. The polar decomposition has multiple applications, such as aerospace computations [5] and factor analysis [35], and can be used as a pre-processing step to calculate the singular value decomposition (SVD) of a general matrix or to solve the eigenvalue problem of a Hermitian matrix [31]. While the cost of QDWH in floating point operations (flops) is high compared to other PD algorithms such as the SVD-based PD, it is based on highly parallel kernels, and so can take better advantage of the available parallel hardware. A synergistic set of advanced dense linear algebra kernels is used to implement the QDWH algorithm, namely QR and Cholesky factorizations and solvers, matrix-matrix multiplication, matrix norm estimation, and condition number estimation. To this end, we rely on the SLATE library [13] to employ the asynchronous task-based hybrid MPI + OpenMP programming paradigm to maximize performance. SLATE uses OpenMP to seamlessly build the directed acyclic graph (DAG) of tasks, pipelines tasks, and track data dependencies during the iterative QDWH algorithm. SLATE employs proven techniques in dense linear algebra, such as a 2D block cyclic data distribution and communication-avoiding algorithms, as well as modern parallel programming approaches, such as dynamic scheduling and communication overlapping. In addition, it employs lookahead techniques to further expose parallelism while actively pursuing the critical path.

We conduct a comprehensive performance analysis using ill-conditioned matrices on distributed memory multi-GPU systems and assess the scalability of the QDWH-based polar decomposition using SLATE. The benchmarking campaign reveals up to an 18× performance speedup using hardware accelerators against the existing state-of-the-art CPU-only implementation, demonstrating the extreme scale applicability of our implementation.

The remainder of the paper is organized as follows. Section 2 highlights the contributions of this paper. Section 3 provides a literature survey on the latest polar decomposition developments, while Section 4 briefly recalls the QDWH-based PD algorithm. An overview of SLATE is in Section 5. Section 6 explains our massively parallel implementation of the resulting task-based QDWH-PD algorithm using SLATE, while Section 7 demonstrates the numerical robustness and reports on the thorough performance benchmarking and profiling campaigns. We conclude in Section 8.

2 CONTRIBUTIONS

Our main contributions are:

- We deploy the first distributed and GPU-accelerated QDWH-based PD algorithm. This is implemented in the open source SLATE library.
- We develop the first QDWH-based PD implementation that supports all four standard data types: float, float complex, double, and double complex; and that supports rectangular matrices ($m \geq n$).
- We implement a matrix 2-norm estimator based on power iteration and a condition number estimator for general and triangular matrices.
- We conduct performance comparisons against the state-of-the-art ScaLAPACK implementation of QDWH-based PD from POLAR [39], and demonstrate the performance scalability on up to 32 nodes and 192 GPUs.
- We demonstrate portability across NVIDIA CUDA and AMD HIP GPU architectures. SLATE also supports SYCL for Intel GPUs on the upcoming *Aurora* system.

3 RELATED WORK

Polar decomposition (PD) algorithms have been studied over the last four decades, with theoretical analysis providing error bounds that guarantee a high degree of accuracy [7, 12, 20, 21, 23, 24]. A framework to compute the SVD and eigenvalue decomposition (EVD) based on the polar decomposition has been suggested by Higham and Papadimitriou [18, 19]. The main steps to compute the SVD through the polar decomposition starts by finding the polar decomposition $A = U_pH$, then the EVD of $H = V\Lambda V^\top$, therefore, $A = (U_pV)\Lambda V^\top = U\Lambda V^\top$, where U and V are the singular vectors and Λ contains the singular values. Moreover, a light-weight version of the polar decomposition can be employed to extract the most significant singular values/vectors [26] and the negative eigen values/vectors [36].

Conversely, the polar decomposition can be derived from the SVD [15, 42] as follows:

$$A = U\Sigma V^\top = UV^\top V\Sigma V^\top = U_pV\Sigma V^\top = U_pH.$$

The PD can also be computed utilizing an iterative approach such as Newton's method, but this can result in numerical instability due to the need for an explicit matrix inversion at each step. An algorithm based on Halley's iteration (DWH) [22, 28] was developed with a cubic rate of convergence in arriving at the final polar factor but, like Newton's method, involves matrix inversions at every iteration.

To solve the numerical accuracy issues due to the matrix inversion, an inverse-free QR-based dynamically-weighted Halley algorithm (QDWH) was proposed by Nakatsukasa et. al [29, 30]. The QDWH-based PD algorithm relies on compute-bound matrix operations, which makes this algorithm a unique candidate to highly utilize the underlying hardware architecture. The QDWH-based PD algorithm has been implemented on both shared memory and distributed memory systems. The first high-performance implementation of the QDWH-based polar decomposition was demonstrated on a shared-memory system with multiple GPUs [41] using the MAGMA library [27], as an accelerated implementation of LAPACK [3]. This implementation adopts the fork-join paradigm. A distributed memory implementation of the QDWH-based PD was later provided by the POLAR library [39], with further improvements in performance based on a topology-aware grid of processors [37]. The high performance implementation of the QDWH-based PD presented in POLAR is based on the ScaLAPACK library [6], which is an extension of LAPACK for distributed memory systems, inheriting the fork-join paradigm. POLAR has been fully integrated into the Cray Scientific and Math Libraries (LibSci) [9] since v17. The Elemental library [34] provides another distributed memory implementation of the QDWH-based PD. The Elemental numerical library is a framework for highly efficient implementations of dense and sparse matrix algorithms. Elemental utilizes an elemental distribution, which fixes the distribution block size to one, thus making it distinct from the algorithmic block size.

Previous work [37] demonstrated that the POLAR QDWH implementation for the polar decomposition outperforms the SVD-based implementation by up to 5× on ill-conditioned matrices. Additionally, POLAR achieved up to 4× speedup compared to the Elemental implementation of the QDWH [37].

The POLAR implementation is limited in terms of its high concurrency, due to the lookahead techniques being impractical with the bulk synchronous fork-join paradigm used in the ScaLAPACK library. This may lead to reduced hardware occupancy, particularly in a strong scaling mode of operation. To address this bottleneck, the first asynchronous task-based QDWH-based PD implementation [38] was presented, achieving improved performance on various shared-memory systems. This implementation, based on tile algorithms [1, 2, 8], used the Chameleon [1] library, which relies on the StarPU [4] dynamic runtime system to pipeline and track the data dependencies of the various fine-grained tasks on homogeneous and heterogeneous architectures. Although StarPU runs on distributed memory environments, the QDWH implementation in Chameleon is limited to a single node. This was due to limitations in StarPU's sequential task flow (STF) programming model to efficiently support collective communications.

Later, another asynchronous task-based implementation of the QDWH-based PD algorithm on distributed memory architectures was introduced [40], based on the DPLASMA library using the PaRSEC dynamic runtime system [10]. PaRSEC utilizes a Domain Specific Language (DSL) called Job Data Flow (JDF) to express the algorithm as a Parameterized Task Graph (PTG), which allows for expressing collective communications directly with the language. This PD implementation runs on CPU architectures exclusively; it is not GPU accelerated. Unfortunately, due to significant changes in the PaRSEC runtime, this research implementation of PD no longer works with the latest PaRSEC, which is required to work on Summit and Frontier.

4 QDWH-BASED POLAR DECOMPOSITION

We rely on the inverse-free QDWH-based iterative procedure [28, 30]. to calculate the polar factors U_pH. To make this paper algorithmically self-contained, we first review the practical QDWH iterative algorithm. Initialize

$$A_0 = A/\|A\|_2\,,$$

then iterate until A_k converges:

$$\begin{aligned}\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R &= \begin{bmatrix} \sqrt{c_k}A_k \\ I \end{bmatrix}, \\ A_{k+1} &= \frac{b_k}{c_k}A_k + \frac{1}{\sqrt{c_k}}\left(a_k - \frac{b_k}{c_k}\right)Q_1Q_2^\top,\end{aligned} \tag{1}$$

for scalars a_k, b_k, c_k as defined in Algorithm 1. At convergence, $U_p = A_{k+1}$ and $H = U_p^T A$. When A_k becomes well-conditioned, it is possible to replace Eq. (1) with a Cholesky-based iteration variant as follows:

$$\begin{aligned} W_k &= \mathrm{chol}(Z_k),\ Z_k = I + c_k A_k^\top A_k, \\ A_{k+1} &= \frac{b_k}{c_k}A_k + \left(a_k - \frac{b_k}{c_k}\right)(A_k W_k^{-1} W_k^{-\top}).\end{aligned} \tag{2}$$

The main computational stages in the QDWH-based PD algorithm are summarized as follows: (1) Estimate the condition number, which can be computed either using the *LU* factorization followed by a condition number estimator, or by using the *QR* factorization followed by a condition number estimator of the upper triangular matrix *R*. (2) The QR-based QDWH iteration, which requires the *QR* factorization as well as building the orthogonal matrix *Q*. (3) The Cholesky-based QDWH iteration. (4) The calculation of the symmetric positive semi-definite polar factor *H*.

The iterative QDWH-based PD procedure relies primarily upon communication-friendly and compute-intensive matrix operations, such as QR and Cholesky factorizations and matrix-matrix multiplication, which make the QDWH-based PD formulation an attractive and practical algorithm to implement. This is in contrast to an SVD-based algorithm, where it is challenging to remove memory-bound Level 2 BLAS operations, and data dependencies prevent a lookahead technique to overlap communication and computation [11].

The number of iterations is determined by the matrix condition number estimation. The theoretical analysis in [30, 31] shows that the upper-bound for the number of iterations is six, assuming double precision arithmetic.

The overall algorithmic complexity of the QDWH-based PD, assuming square matrices for simplicity, is given by:

$$\tfrac{4}{3}n^3 + (8 + \tfrac{2}{3})n^3 \times \#it_{QR} + (4 + \tfrac{1}{3})n^3 \times \#it_{Chol} + 2n^3,$$

where $\#it_{QR}$ and $\#it_{Chol}$ are the number of QR-based and Cholesky-based iterations, respectively. Experimentally, testing ill-conditioned matrices requires three QR and three Cholesky-based iterations, while well-conditioned matrices need two Cholesky-based and no QR-based iterations.

5 SLATE: SOFTWARE FOR LINEAR ALGEBRA TARGETING EXASCALE

SLATE delivers fundamental dense linear algebra capabilities for current and upcoming distributed memory systems, including GPU-accelerated systems as well as more traditional multi-core CPU-only systems. SLATE provides coverage of existing ScaLAPACK functionality, including parallel implementations of Basic Linear Algebra Subroutines (BLAS), linear systems solvers, least squares solvers, and singular value and eigenvalue solvers, plus new algorithms such as the polar decomposition. In this respect, SLATE serves as a replacement for ScaLAPACK, which, after almost three decades of operation, cannot be adequately retrofitted for modern, GPU-accelerated architectures.

SLATE is built on top of the C++17, MPI, and OpenMP standards, as well as de facto industry standard solutions such as NVIDIA CUDA, AMD HIP, and SYCL, with the goal to create a portable, high-performance library [14]. SLATE also relies on high performance implementations of numerical kernels from vendor libraries, such as Intel MKL, IBM ESSL, NVIDIA cuBLAS, and AMD rocBLAS. SLATE interacts with these libraries through the BLAS++ and LAPACK++ portability layer. SLATE aims to extract the full performance potential and maximum scalability from modern, many-node HPC machines with large numbers of cores and multiple hardware accelerators per node.

SLATE can use GPU-aware MPI if available to directly communicate tiles from one GPU's memory across the network to another GPU's memory. This has proved especially beneficial on Frontier, the first exascale class machine operated by the U.S. Dept. of Energy, where the network interface card (NIC) is attached to the GPUs rather than the CPUs.

6 HIGH PERFORMANCE IMPLEMENTATION

6.1 Implementation Details

Algorithm 1 shows the computational stages of QDWH PD using SLATE. First, an estimation of the matrix two-norm is required to scale the matrix, which is essential to improve the convergence rate (lines 11 to 13). Then, the condition number is estimated based on the *QR* factorization of the matrix (lines 15 to 19). We implement `norm2est` and `trcondest` in SLATE to provide these matrix metrics. After that, lines 22 to 50 show the main computational steps to compute the orthogonal polar factor *U*. At each iteration, the parameters a, b, c, Li are updated. Based on the variable c at line 29, which is derived from the condition number of the matrix, the loop enters either the QR-based iteration at line 30, or the Cholesky-based iteration at line 39. This algorithmic change to using a Cholesky-based iteration when the matrix becomes well-conditioned has lower algorithmic

Algorithm 1 Pseudo-code for the QDWH-based PD algorithm.

```
1:  procedure QDWH( Matrix $A \in \mathbb{C}^{m\times n}$, Matrix $H \in \mathbb{C}^{n\times n}$ )
2:      // On output, $A$ is overwritten by unitary matrix $U_p$
3:      // and $H$ is positive semidefinite matrix
4:      // Allocate distributed matrix workspaces
5:      Matrix $W = \begin{bmatrix} W_1 \\ W_2 \end{bmatrix}, W_1 \in \mathbb{C}^{m\times n}, W_2 \in \mathbb{C}^{n\times n}$
6:      Matrix $Q = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}, Q_1 \in \mathbb{C}^{m\times n}, Q_2 \in \mathbb{C}^{n\times n}$
7:      // Backup A in Acpy to compute H
8:      copy( $A$, $A$cpy )
9:      Anorm = norm( Norm::One, $A$ )                              ▷ $\|A\|_1$
10:     // Estimate the two-norm.
11:     $\alpha$ = norm2est( $A$ )                                  ▷ $\alpha \approx \|A\|_2$
12:     // $A$ stores $A_k$, which converges to $U_p$
13:     scale( 1./$\alpha$, $A$ )                                   ▷ $A_0 = A/\alpha$
14:     // Estimate the condition number
15:     copy( $A$, $W_1$ )
16:     geqrf( $W_1$ )                                              ▷ $A = QR$ factorization
17:     Rcondest = trcondest( $W_1$ )                               ▷ $\approx \mathrm{cond}_1(R)$
18:     Rnorm = norm( $W_1$ )                                       ▷ $\approx \|R\|_1$
19:     $l_0$ = Anorm*Rcondest $/\sqrt{n}$
20:     // The polar decomposition iterations
21:     $k = 1,\ L_i = l_0,$ conv = 100
22:     while (conv $\geq \sqrt[3]{5\epsilon}$ or $|L_i - 1| \geq 5\epsilon$) do
23:         $L_2 = L_i^2,\ dd = \sqrt[3]{(4(1-L_2)/L_2^2)}$
24:         $sqd = \sqrt{1+dd}$
25:         $a1 = sqd + \sqrt{8 - 4\times dd + 8(2-L_2)/(L_2 \times sqd))}/2$
26:         $a = \mathrm{real}(a1);\ b = (a-1)^2/4;\ c = a+b-1$
27:         $L_i = L_i(a + b\times L_2)/(1+cL_2)$
28:         // Compute $A_k$ from $A_{k-1}$
29:         if $c > 100$ then
30:             set $W = \begin{bmatrix} W_1 \\ W_2 \end{bmatrix} = \begin{bmatrix} \sqrt{c}A_{k-1} \\ I \end{bmatrix}$
31:             geqrf( $W$ )                                        ▷ $W = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R$ factorization
32:             unmqr( $W$, $Q$ )                                   ▷ generate $Q$ explicitly
33:             copy( $A$, $W_1$ )                                  ▷ Save $W_1 = A_{k-1}$ for conv
34:             $\theta = \frac{1}{\sqrt{c}}\left(a - \frac{b}{c}\right);\ \beta = \frac{b}{c}$
35:             gemm( $\theta$, $Q_1$, $Q_2^\top$, $\beta$, $A$ )
36:                                  ▷ $A_k = \frac{1}{\sqrt{c}}\left(a-\frac{b}{c}\right)Q_1Q_2^\top + \frac{b}{c}A_{k-1}$
37:         else
38:             copy( $A$, $W_1$ )                                  ▷ Save $W_1 = A_{k-1}$
39:             set $W_2$ = Identity
40:             herk( $-c$, $A$, one, $W_2$ )                       ▷ $W_2 = I - cA_{k-1}^\top A_{k-1}$
41:             posv( $W_2$, $A^\top$ )                             ▷ Solve $W_2X = A_{k-1}^\top$,
42:                                                                 ▷ $X$ overwrites $A$
43:             $\theta = \frac{b}{c};\ \beta = \left(a - \frac{b}{c}\right)$
44:             add( $\theta$, $W_1$, $\beta$, $A$ )
45:                                  ▷ $A_k = \frac{b}{c}A_{k-1} + \left(a - \frac{b}{c}\right)(A_{k-1}W_2^{-1}W_2^{-T})$
46:         end if
47:         add( one, $A$, $-$one, $W_1$ )                          ▷ $W_1 = A_k - A_{k-1}$
48:         conv = norm( Norm::Fro, $W_1$ )                         ▷ $\|A_k - A_{k-1}\|_F$
49:         $k = k+1$
50:     end while
51:     // Compute $H$
52:     gemm( one, $A$, $A$cpy, zero, $H$ )                         ▷ $H = U_p^\top A$
53: end procedure
```

complexity than the QR-based iteration, while still maintaining numerical stability. The convergence tolerance used in QDWH line 22 and for other cubically convergent methods are summarized in [29]. Finally, the symmetric positive semi-definite polar factor H is computed in line 52, after exiting the main computational loop.

6.2 The Two Norm Estimation

The two-norm matrix estimator identifies the largest singular value of the matrix. We employed the power iteration method [17] as our approach to estimate the two-norm of the matrix. Algorithm 2 presents our implementation of the two-norm matrix estimator `norm2est` in SLATE. The initial vector to start the computational loop consists of the sum of each matrix column, which is computed on line 6. We call `internal::norm` to calculate the column sums that are local to each processor, then we utilize MPI Allreduce to find the global column sums. The iterations involve the multiplication of the matrix A and $A^\top$ by vectors as shown in lines 18 and 19. To carry out the matrix-vector multiplication involved in `norm2est`, we develop `gemmA`, which is a variant of `gemm` that optimizes the data movements and performance when the A matrix is large relative to the C matrix. To minimize the amount of data movement, tiles of B are sent to where the tiles of A reside to compute partial results using tile multiplications, then the final result is computed by performing a parallel reduction to where the output C tiles reside. The convergence tolerance used to estimate the matrix two norm is 0.1, where approximations accurate to a factor of 5, for example, are entirely satisfactory, and QDWH converges within 6 iterations as explained in [31].

6.3 Condition Number Estimate

Estimating the condition number of matrix is an important step towards determining the number of iterations required for convergence, as well as calculating the a, b, c parameters that help to achieve cubic convergence during the QDWH iteration. In particular, the value of c determines whether a QR-based or Cholesky-based needs to be performed during the iterative loop. The reciprocal of the 1-norm condition number estimate of matrix A is rcond $= \frac{1}{\|A\|_1 \times \|A^{-1}\|_1}$. We implemented `norm1est` to approximate $\|A^{-1}\|_1$ based on Hager's algorithm [16]. As in (Sca)LAPACK, the algorithm uses reverse communication to evaluate matrix-vector products and solves, allowing for a single implementation of the 1-norm estimate to evaluate the condition number for any possible matrix factorization. We provide `gecondest` to compute the condition number of a matrix given its LU factorization, and `trcondest` to compute the condition number of a triangular matrix. In QDWH, we call `trcondest` on the triangular matrix R from $A = QR$.

7 NUMERICAL RESULTS AND ANALYSIS

7.1 Environment Settings

Our experiments were conducted on the IBM POWER9 system *Summit* and the HPE Cray EX system *Frontier* at Oak Ridge National Laboratory. *Summit* has 4,608 compute nodes; each node contains two 22-core IBM POWER9 CPUs and 6 NVIDIA Volta V100 GPUs, connected by NVIDIA's high-speed NVLink, with 512 GiB DDR4 CPU RAM and 96 GiB high bandwidth memory (HBM2)

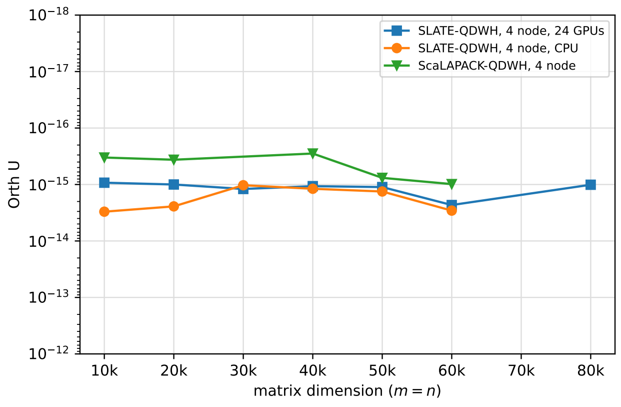
Algorithm 2 Pseudo-code for the two-norm estimation.

```
function NORM2EST( Matrix $A \in \mathbb{C}^{m\times n}$ )
    // Create SLATE distributed vectors
    Vector $X \in \mathbb{C}^n$
    Vector $AX \in \mathbb{C}^m$
    // Compute local sum of all columns
    internal::norm( Norm::One, A, local_sum )
    // Reduce local sum to get global sum of all cols in $X$
    MPI_Allreduce( local_sum, $X$, $n$, ... )
    // Compute initial estimate $e$
    $e$ = norm( Norm::Fro, $X$ )
    $e_0 = 0$
    normX = $e$
    tol = 0.1
    while $|e - e0| > \text{tol}*e$ do
        $e_0 = e$
        scale( 1/normX, $X$ )
        // Compute Ax = A * sx
        gemmA( one, $A, X$, zero, $AX$ )
        gemmA( one, $A^\top, AX$, zero, $X$ )
        // Update $e$
        normX = norm( Norm::Fro, X )
        normAX = norm( Norm::Fro, AX )
        $e$ = normX / normAX
    end while
    return $e$
end function
```

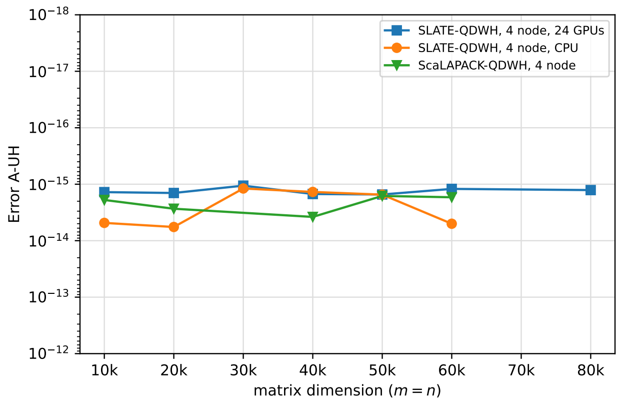
GPU RAM per node. Two cores per node are reserved for the OS. Nodes are connected by a dual-rail Mellanox EDR 100G InfiniBand network [33]. Executables were generated with the GNU g++ compiler and IBM Spectrum MPI. We used gcc v9.1.0, cuda v11.5.2, essl v6.3.0 and spectrum-mpi v10.4.0.3-20210112 to compile and run tests on *Summit*.

Frontier has 9,408 compute nodes; each node consists of a 64-core AMD Optimized 3rd Gen EPYC CPU with 512 GiB of DDR4 memory and 4 AMD MI250X GPUs, each with 2 Graphics Compute Dies (GCDs) for a total of 8 GCDs per node. Each GCD has 64 GiB of high-bandwidth memory (HBM2E). 8 cores per node are reserved for the OS. The CPU is connected to each GCD via Infinity Fabric, with a peak bandwidth of 36 GB/s in each direction. The GCDs are connected by Infinity Fabric with varying number of links, yielding bandwidths of 50, 100, or 200 GB/s between GCDs [32]. We used PrgEnv-gnu v8.3.3, cray-mpich v8.1.23 and rocm v5.3.0 to compile and test on *Frontier*.

All experiments were conducted with IEEE double-precision arithmetic. We implemented a matrix generator to test various synthetic matrices with different condition numbers and distributions of singular values. The condition number has the most significant effect on the convergence of QDWH and, consequently, its performance. The theoretical analysis in [31] proved that the maximum number of iterations for convergence is six for ill-condition matrices in double precision arithmetic. For these experiments, the generator creates random unitary matrices U, V, obtained through the QR factorization of random matrices, and a diagonal matrix Σ based on the desired condition number of the matrix A. It then multiplies these together, forming $A = U\Sigma V^H$ from its SVD. We



(a) Orthogonality error of U_p.



(b) Backward error of the polar decomposition.

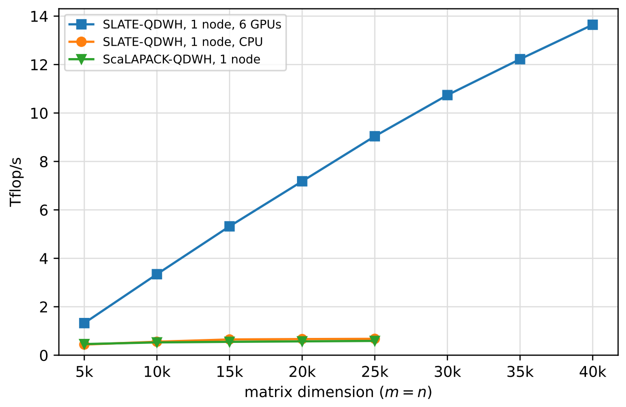
Figure 1: Accuracy assessment of SLATE and ScaLAPACK QDWH.

generated ill-conditioned matrices with condition number $\kappa = 10^{16}$. In this case, QDWH requires the maximum number of iterations.

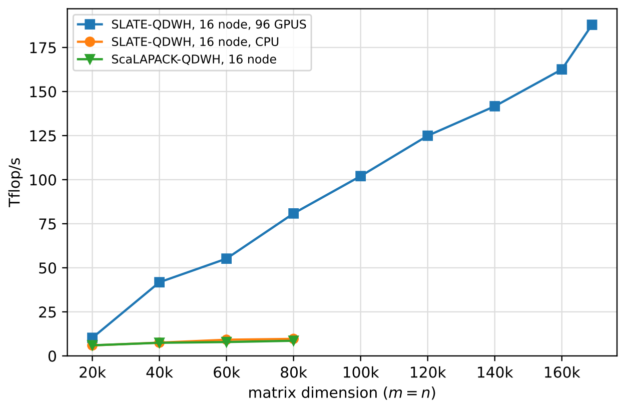
We compare against the ScaLAPACK reference implementation of QDWH provided by POLAR. On Summit, all ScaLAPACK runs used 1 MPI rank per CPU core (42 per node), while SLATE runs used 2 MPI ranks and 6 V100 GPUs per node (3 per MPI rank). On Frontier, all ScaLAPACK runs used 1 MPI rank per CPU core (56 per node), while SLATE runs used 8 MPI ranks and 8 MI250X GCDs per node (1 per MPI rank).

7.2 Performance Results and Analysis

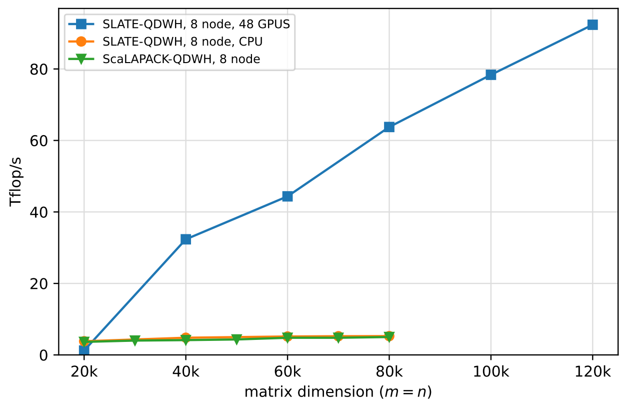
We tested ill-conditioned matrices, which represents the worst-case scenario, where the number of iterations required for convergence is a maximum of six (three QR-based, three Cholesky-based). We first check on the accuracy of our new formulation of QDWH-based PD. For the orthogonality error of U_p, we use: $\frac{\left\|I-U_p^\top U_p\right\|_F}{\sqrt{n}}$, and for the accuracy of the computed polar decomposition U_pH, we use



(a) 1 node of Summit (42 Power9 CPUs 6 V100 GPUs)

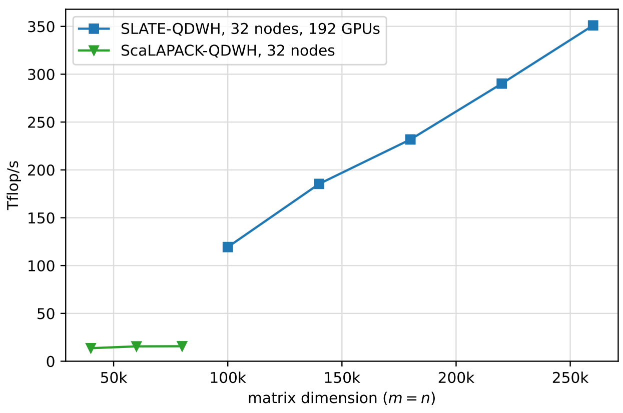


(a) 16 nodes of Summit (672 Power9 CPUs, 96 V100 GPUs).



(b) 8 nodes of Summit (336 Power9 CPUs, 48 V100 GPUs).

Figure 2: Performance comparison on Summit.



(b) 32 nodes of Summit (1344 Power9 CPUs, 192 V100 GPUs).

Figure 3: Performance comparison on Summit.

the backward error: $\frac{\|A-U_pH\|_F}{\|A\|_F}$. Fig. 1a displays the orthogonality error of the computed polar factor U_p, and Fig. 1b presents the backward error of the overall polar decomposition, which remain around machine precision (i.e., 10^{-15}) for both the SLATE and ScaLAPACK implementations of QDWH for matrices of various sizes. This highlights the numerical stability of the new task-based formulation of QDWH.

We executed several tests to find the best tile size n_b to deliver the highest performance. For SLATE-QDWH tests on GPUs, the tuning tests we conducted showed that a tile size of n_b = 320 provided the best performance compared to other tested tile sizes. For tests on CPUs, n_b = 192 gave the best performance among other tested tile sizes.

Figs. 2 and 3 compare the performance in Tflop/s of the SLATE and ScaLAPACK implementations of QDWH-based PD across various number of nodes on Summit. The GPU-accelerated SLATE implementation of QDWH (blue squares) outperforms its counterpart implementation in ScaLAPACK (green triangles), and the performance gap widens as the matrix size increase. SLATE-QDWH is faster by up to 18× on 1 nodes and 4 nodes, and by approximately 13× on 8 nodes. The SLATE-QDWH performance on GPUs grows as the matrix size increases, where larger matrices can highly exploit the GPUs. Using only CPU cores, SLATE's performance (orange circles) is similar to the ScaLAPACK performance (green triangles). Due to time constraints, we limited the size of CPU runs once the peak performance was evident. Fig. 4 displays the SLATE-QDWH performance with various number of Summit nodes. While the strong scalability for a fixed problem size is limited, it achieves good weak scalability at the largest problem size for each number of nodes.

Fig. 6 shows the scalability performance of SLATE QDWH using several number of nodes on Frontier. The SLATE QDWH performance increases as the number of nodes increases and as the matrix size grows. The task-based SLATE-QDWH using GPUs achieves around 180 Tflop/s on 16 nodes equipped with 128 GPUs, approximately 24% of the peak performance. The maximum matrix size that can be tested on this number of nodes is 175k, due to the large memory footprint of the algorithm, as discussed in [37]. Additionally, data movement between nodes and between CPU and GPU are

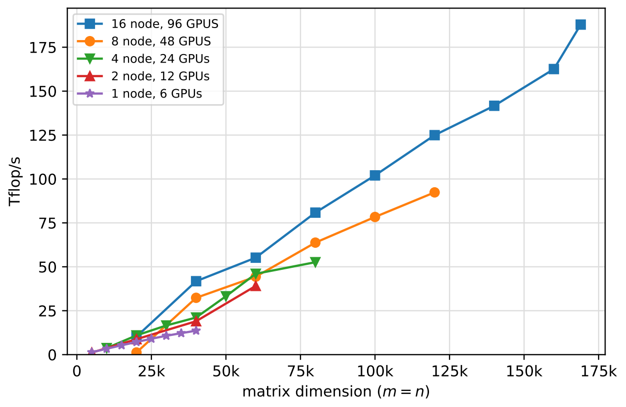


Figure 4: Scalability study of SLATE QDWH on various number of Summit nodes.

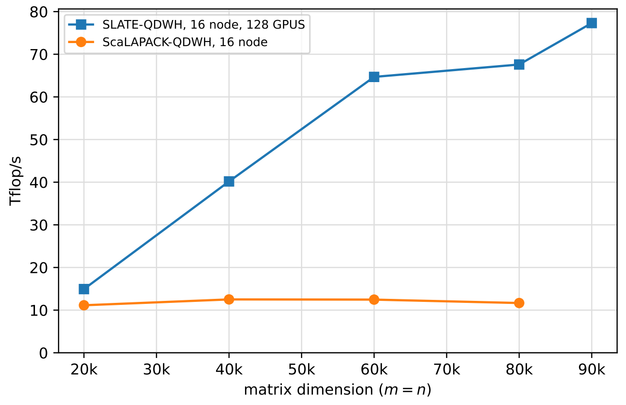


Figure 5: 16 nodes of Frontier (896 EPYC CPUs, 128 MI250X GPUs).

further performance limiting factors. GPU-aware MPI can mitigate the performance impact caused by the data movements. SLATE benefits from GPU-aware MPI on Frontier, where the NICs are attached to the GPUs. However, on Summit the NICs are attached to the CPUs, so MPI communication always goes through the CPU, explicitly or implicitly, thus GPU-aware MPI is not beneficial for SLATE there.

8 CONCLUSION AND FUTURE WORK

We have introduced a novel task-based implementation of QDWH-based PD on massively parallel systems enhanced with multiple GPUs. We rely on SLATE to employ modern techniques in our formulation of QDWH-based PD, such as communication-avoiding algorithms, lookahead panels to overlap communication and computation, and task-based scheduling, along with a modern C++ framework. This implementation of QDWH in SLATE provides

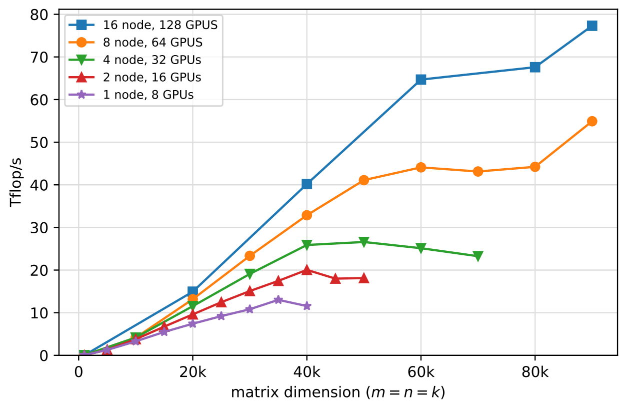


Figure 6: Scalability study of SLATE QDWH on various number of Frontier nodes.

a portable code that can be used on various hardware architectures, including CPUs and NVIDIA, AMD, and Intel GPUs. The benchmarking experiments show that the new task-based high performance implementation of the QDWH-based PD using SLATE outperforms by up to 18× the POLAR ScaLAPACK-QDWH implementation.

For future work, we would like to use QDWH polar decomposition as the main building block to develop partial EVD implementations, to support more economical partial spectrum requirements. Moreover, we would like to integrate mixed-precision techniques to further accelerate the polar decomposition computations. Last but not least, we would like to extend this work to implement another QDWH algorithmic variant, the Zolo PD algorithm [25], which requires an even higher number of flops than QDWH-based PD, but can exploit a higher level of concurrency, making it attractive in the strong-scaling regime.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] 2018. The Chameleon Project. http://project.inria.fr/.
[2] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, Vol. 180.
[3] Edward Anderson, Zhaojun Bai, Christian Heinrich Bischof, Laura Susan Blackford, James Weldon Demmel, Jack J Dongarra, Jeremy J Du Croz, Anne Greenbaum, Sven Hammarling, A McKenney, and Danny C Sorensen. 1999. *LAPACK User's Guide* (3rd ed.). Society for Industrial and Applied Mathematics, Philadelphia.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[5] I. Bar-Itzhack. 1975. Iterative Optimal Orthogonalization of the Strapdown Matrix. *IEEE Transactions on Aerospace Electronic Systems* 11 (Jan. 1975), 30–37. https://doi.org/10.1109/TAES.1975.308025

[6] L. Suzan Blackford, J. Choi, Andy Cleary, Eduardo F. D'Azevedo, James W. Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clint Whaley. 1997. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia.

[7] Ralph Byers and Hongguo Xu. 2008. A New Scaling for Newton's Iteration for the Polar Decomposition and its Backward Stability. *SIAM J. Matrix Anal. Appl.* 30, 2 (2008), 822–843. http://dx.doi.org/10.1137/070699895

[8] Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. 2007. Supermatrix out-of-order scheduling of matrix operations for SMP and multicore architectures. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures* (San Diego, California, USA). ACM, New York, NY, USA, 116–125. https://doi.org/10.1145/1248377.1248397

[9] Cray. [n. d.]. *LibSci.* http://docs.cray.com

[10] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. 2014. PTG: An abstraction for unhindered parallelism. *Proceedings of WOLFHPC 2014: 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Stor* (2014), 21–30. https://doi.org/10.1109/WOLFHPC.2014.8

[11] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. 2018. The Singular Value Decomposition: Anatomy of Optimizing an Algorithm for Extreme Scale. *SIAM Rev.* 60, 4 (2018), 808–865. https://doi.org/10.1137/17M1117732

[12] Walter Gander. 1985. On Halley's iteration method. *Amer. Math. Monthly* 92, 2 (1985), 131–134.

[13] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 26, 18 pages. https://doi.org/10.1145/3295500.3356223

[14] Mark Gates, Asim YarKhan, Dalal Sukkari, Kadir Akbudak, Sebastien Cayrols, Daniel Bielich, and Ahmad Abdelfattah. 2022. Portable and Efficient Dense Linear Algebra in the Beginning of the Exascale Era. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 36–46. https://doi.org/10.1109/P3HPC56579.2022.00009

[15] Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations* (third ed.). Johns Hopkins University Press, Baltimore, Maryland.

[16] William W. Hager. 1984. Condition Estimates. *SIAM J. Sci. Statist. Comput.* 5, 2 (1984), 311–316. https://doi.org/10.1137/0905023 arXiv:https://doi.org/10.1137/0905023

[17] Nicholas J. Higham. 1992. Estimating the matrix p-norm. *Numer. Math.* 62, 1 (01 Dec. 1992), 539–555. https://doi.org/10.1007/BF01396242

[18] Nicholas J. Higham and Pythagoras Papadimitriou. 1993. *Parallel Singular Value Decomposition via the Polar Decomposition.* Numerical Analysis Report No. 239. University of Manchester, England. ftp://vtx.ma.man.ac.uk/pub/narep/narep239.dvi.Z

[19] Nicholas J. Higham and Pythagoras Papadimitriou. 1994. A New Parallel Algorithm for Computing the Singular Value Decomposition. In *Proceedings of the Fifth SIAM Conference on Applied Linear Algebra*, John G. Lewis (Ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 80–84.

[20] Nicholas J. Higham and Pythagoras Papadimitriou. 1994. A Parallel Algorithm for Computing the Polar Decomposition. *Parallel Comput.* 20, 8 (Aug. 1994), 1161–1173.

[21] Charles Kenney and Alan J. Laub. 1992. On Scaling Newton's Method for Polar Decomposition and the Matrix Sign Function. *SIAM J. Matrix Anal. Appl.* 13, 3 (1992), 688–706. https://doi.org/10.1137/0613044 arXiv:http://dx.doi.org/10.1137/0613044

[22] Charles Kenney and Alan J. Laub. 1992. On Scaling Newton's Method for Polar Decomposition and the Matrix Sign Function. *SIAM J. Matrix Anal. Appl.* 13, 3 (1992), 688–706. https://doi.org/10.1137/0613044 arXiv:https://doi.org/10.1137/0613044

[23] Andrzej Kielbasinski and Krystyna Zietak. 2003. Numerical Behaviour of Higham's Scaled Method for Polar Decomposition. *Numerical Algorithms* 32, 2-4 (2003), 105–140. http://dx.doi.org/10.1023/A:1024098014869

[24] B. Laszkiewicz and K. Zietak. 2006. Approximation of Matrices and a Family of Gander Methods for Polar Decomposition. *BIT Numerical Mathematics* 46, 2 (2006), 345–366. http://dx.doi.org/10.1007/s10543-006-0053-4

[25] Hatem Ltaief, Dalal Sukkari, Aniello Esposito, Yuji Nakatsukasa, and David Keyes. 2019. Massively Parallel Polar Decomposition on Distributed-Memory Systems. *ACM Trans. Parallel Comput.* 6, 1, Article 4 (jun 2019), 15 pages. https://doi.org/10.1145/3328723

[26] Hatem Ltaief, Dalal Sukkari, Oliver Guyon, and David Keyes. 2018. Extreme Computing for Extreme Adaptive Optics: The Key to Finding Life Outside Our Solar System. In *PASC 2018: Proceedings of the Platform for Advanced Scientific Computing Conference* (Basel, Switzerland). ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/3218176.3218225 Best Paper.

[27] MAGMA. 2009. Matrix Algebra on GPU and Multicore Architectures. Innovative Computing Laboratory, University of Tennessee. Available at http://icl.cs.utk.edu/magma/.

[28] Yuji Nakatsukasa, Zhaojun Bai, and François Gygi. 2010. Optimizing Halley's Iteration for Computing the Matrix Polar Decomposition. *SIAM J. Matrix Anal. Appl.* (2010), 2700–2720.

[29] Yuji Nakatsukasa, Zhaojun Bai, and François Gygi. 2010. Optimizing Halley's Iteration for Computing the Matrix Polar Decomposition. *SIAM J. Matrix Anal. Appl.* (2010), 2700–2720.

[30] Yuji Nakatsukasa and Nicholas J. Higham. 2012. Backward Stability of Iterations for Computing the Polar Decomposition. *SIAM J. Matrix Anal. Appl.* 33, 2 (2012), 460–479. https://doi.org/10.1137/110857544 arXiv:https://doi.org/10.1137/110857544

[31] Yuji Nakatsukasa and Nicholas J. Higham. 2013. Stable and Efficient Spectral Divide and Conquer Algorithms for the Symmetric Eigenvalue Decomposition and the SVD. *SIAM Journal on Scientific Computing* 35, 3 (2013), A1325–A1349. https://doi.org/10.1137/120876605 arXiv:http://epubs.siam.org/doi/pdf/10.1137/120876605

[32] Oak Ridge Leadership Computing Facility (OLCF). 2023. *Frontier User Guide.* https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

[33] Oak Ridge Leadership Computing Facility (OLCF). 2023. *Summit User Guide.* https://docs.olcf.ornl.gov/systems/summit_user_guide.html

[34] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. 2013. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Software* 39, 2 (2013), 13. http://doi.acm.org/10.1145/2427023.2427030

[35] PeterH. Schönemann. 1966. A generalized solution of the orthogonal Procrustes problem. *Psychometrika* 31, 1 (1966), 1–10. https://doi.org/10.1007/BF02289451

[36] Dalal Sukkari. 2019. *High Performance Polar Decomposition on Manycore Systems and its application to Symmetric Eigensolvers and the Singular Value Decomposition.* Ph. D. Dissertation. KAUST. https://doi.org/10.25781/KAUST-R20B1

[37] Dalal Sukkari, Hatem Ltaief, Aniello Esposito, and David Keyes. 2019. A QDWH-based SVD Software Framework on Distributed-memory Manycore Systems. *ACM Trans. Math. Softw.* 45, 2, Article 18 (April 2019), 21 pages. https://doi.org/10.1145/3309548

[38] D. Sukkari, H. Ltaief, M. Faverge, and D. Keyes. 2018. Asynchronous Task-Based Polar Decomposition on Single Node Manycore Architectures. *IEEE Transactions on Parallel and Distributed Systems* 29, 2 (Feb 2018), 312–323. https://doi.org/10.1109/TPDS.2017.2755655

[39] Dalal Sukkari, Hatem Ltaief, and David Keyes. 2016. High Performance Polar Decomposition on Distributed Memory Systems. In *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9833)*, Pierre-François Dutot and Denis Trystram (Eds.). Springer, 605–616. http://dx.doi.org/10.1007/978-3-319-43659-3

[40] D. Sukkari, H. Ltaief, D. Keyes, and M. Faverge. 2019. Leveraging Task-Based Polar Decomposition Using PARSEC on Massively Parallel Systems. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–12.

[41] Dalal Sukkari, Hatem Ltaief, and David E. Keyes. 2016. A High Performance QDWH-SVD Solver Using Hardware Accelerators. *ACM Trans. Math. Softw* 43, 1 (2016), 6:1–6:25. http://doi.acm.org/10.1145/2894747

[42] Lloyd N. Trefethen and David Bau. 1997. *Numerical Linear Algebra.* SIAM, Philadelphia, PA. http://www.siam.org/books/OT50/Index.htm